Composants logiciels

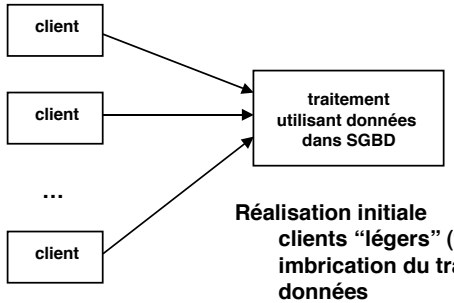
Exemples : *Java Beans*, *Enterprise Java Beans*

Sacha Krakowiak
Université Joseph Fourier
Projet Sardes (INRIA et IMAG-LSR)

http://sardes.inrialpes.fr/~krakowia

Introduction : évolution des architectures client-serveur

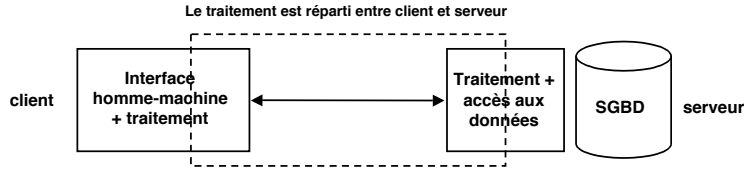
- **Schéma de base d'une application**
  - Accès, depuis des clients, à des données permanentes (SGBD), avec traitement spécifique



Réalisation initiale
clients "légers" (terminaux alphanumérique)
imbrication du traitement et de l'accès aux données

Évolution des architectures client-serveur (2)

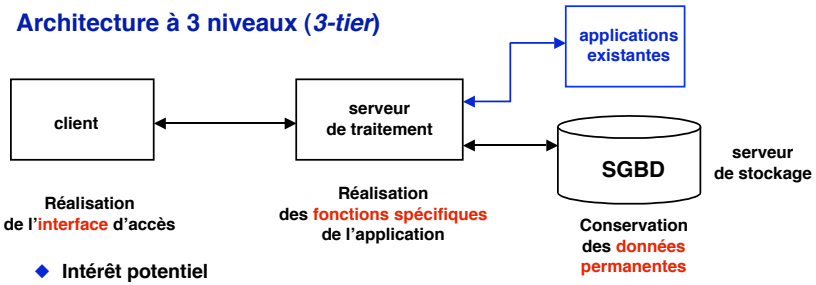
Architecture à 2 niveaux (*2-tier*)
Réservée à des cas simples où tout ou partie du traitement peut être fait chez le client



Limitations
- puissance de traitement sur le poste client
- partage des données
- capacité de croissance
- facilité d'évolution

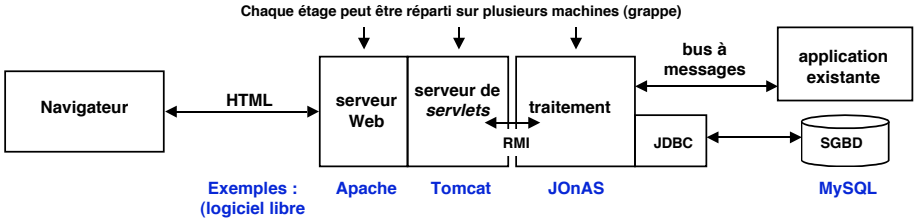
Évolution des architectures client-serveur (3)

- **Architecture à 3 niveaux (*3-tier*)**

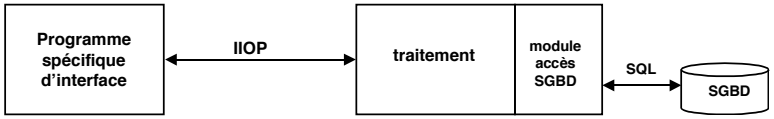


  - Intérêt potentiel
    - séparation des fonctions
    - interfaces bien définies, normalisation, ouverture
    - capacité d'évolution et de croissance
    - réutilisation de l'existant

Variations sur les schémas d'architecture



Répond à un besoin fort : la fourniture de pages Web dynamiques



Réalisation d'une application répartie

- Objectifs
  - Facilité de développement (donc coût réduit)
  - Capacité d'évolution et de croissance
  - Ouverture (intégration, supports hétérogènes, etc)
- Solutions possibles de réalisation
  - Les outils vus jusqu'ici sont utilisables …
    - Java RMI, CORBA, Bus à messages
  - … mais doivent être complétés
    - construction modulaire facilitant l'évolution et l'ouverture
    - services communs (pour ne pas "réinventer la roue" et se concentrer sur l'application spécifique)
    - outils de développement (écriture, assemblage)
    - outils de déploiement (mise en place des éléments)
    - outils d'administration (observation, reconfiguration)
  - Les composants visent à fournir ces compléments

Plan de présentation

- Introduction aux composants logiciels
  - Des objets aux composants
  - Notions d'architecture logicielle
- Un exemple de composants pour clients : *Java Beans*
  - Principes
  - Applications
- Un exemple de composants pour serveurs : *Enterprise Java Beans* (EJB)
  - Principes
  - Applications
- Un exemple d'intégration d'applications à grande échelle: *Web Services*

Les limites de la programmation par objets

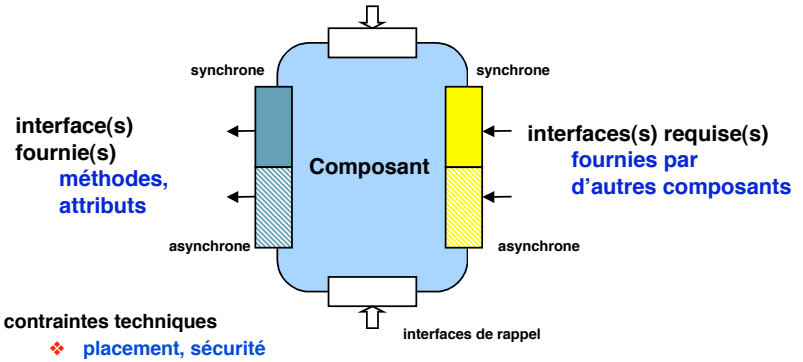
- Pas d'expression des ressources nécessaires
  - Seules sont définies les interfaces fournies, non celles requises
- Absence de vision globale de l'application
  - les principaux concepts sont définis au niveau d'un objet individuel
  - pas de notion de description globale de l'architecture
- Difficulté d'évolution
  - conséquence de l'absence de vision globale
- [pour certaines infrastructures] : absence de services (propriétés "non fonctionnelles")
  - les services nécessaires doivent être réalisés "à la main" (persistance, sécurité, tolérance aux fautes, etc.)
- Absence d'outils d'administration (composition, déploiement)
- Conclusion
  - charge importante pour le programmeur
  - incidence sur la qualité de l'application
  - une partie du cycle de vie n'est pas couverte

Composants : définition

- **Définition**
  - module logiciel autonome
    - unité de déploiement (installation sur différentes plates-formes)
    - unité de composition (combinaison avec d'autres composants)
- **Propriétés**
  - spécifie explicitement la ou les interface(s) fournie(s) (attributs, méthodes)
  - spécifie explicitement la ou les interface(s) requise(s) pour son exécution
  - peut être configuré
  - capable de s'auto-décrire
- **Intérêt**
  - permettre la construction d'applications par composition de briques de base configurables
  - séparer les fonctions des fournisseur et d'assembleur (conditions pour le développement d'une industrie des composants)

Composants : modèle générique

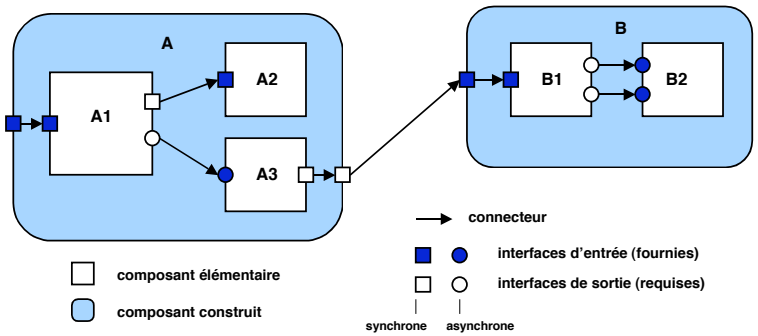
propriétés configurables
interfaces spéciales avec accès restreint



contraintes techniques
- placement, sécurité
- accès transactionnel, persistance
- interfaces fournies par le système (bibliothèques, etc.)

Composants : utilisation

Composition hiérarchique et encapsulation
(composants construits, sous-composants)
Interconnexion de composants
(connecteurs, ou objets de communication)



Support logiciel pour composants

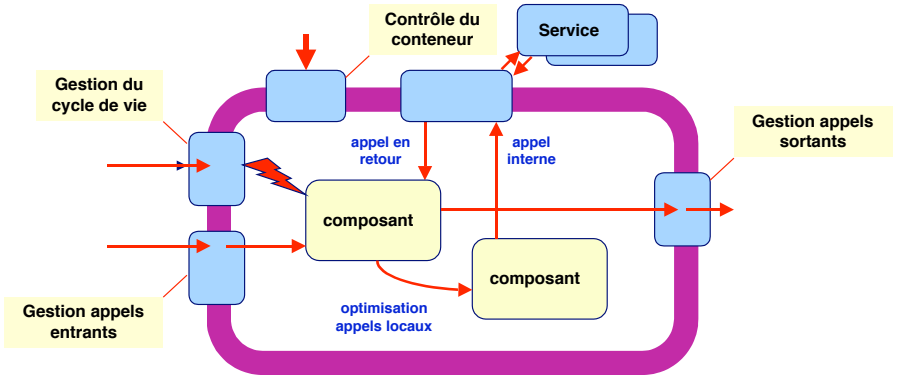
Pour assurer leurs fonctions, les composants ont besoin d'un support logiciel.
On distingue généralement conteneurs et structures d'accueil

- **Conteneur**
  - encapsulation d'un composant
  - prise en charge des services du système
    - nommage, sécurité, transactions, persistance, etc.
  - prise en charge partielle des relations entre composants (connecteurs)
    - invocations, événements
  - techniques utilisées : interposition, délégation
- **Structure d'accueil**
  - espace d'exécution des conteneurs et composants
  - médiateur entre conteneurs et services du système

Conteneur

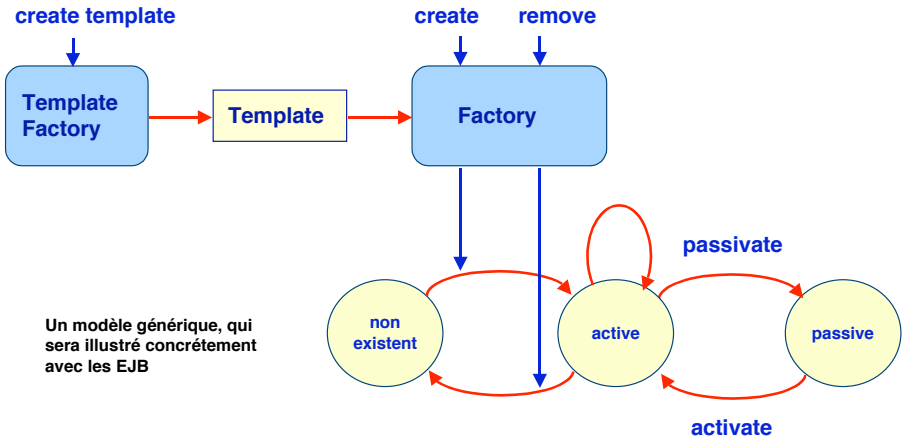
- **Mise en œuvre du contrôleur de composants**
  - Conteneur : infrastructure de gestion pour un ou plusieurs composants
    - Par composant ou par “type”
  - Rappel des fonctions : cycle de vie, médiation, gestion de services, composition (si composants composites)
  - Génération automatique des conteneurs à partir
    - des descriptions d’interfaces des composants (cf. talons)
    - de paramètres de configuration fournis
  - “Contrats” entre le conteneur et les composants inclus
    - Interface interne du contrôleur
    - Interfaces de rappel des composants

Schéma générique de conteneur



Illustrations spécifiques pour différents modèles : EJB, CCM, OSGi, Fractal/Julia, …

Cycle de vie des composants (1)



Un modèle générique, qui sera illustré concrétement avec les EJB

Cycle de vie des composants (2)

- **Mécanismes pour économiser les ressources**
  - Activation-passivation
    - *passivate* : Élimine le composant de la mémoire si non utilisé (en stockant son état) - appelé par le conteneur
    - *activate* : Restitue l’état du composant ,qui peut être réutilisé
  - *Pool* d’instances
    - Le conteneur maintient un *pool* fixe d’instances de composants qui peuvent donner vie à des composants “virtuels”
    - Un composant virtuel incarné par une instance du *pool* devient utilisable, à condition de charger son état => méthodes de rappel pour sauver-restaurer l’état
    - Évidemment plus simple pour les composants sans état
  - Différences entre passivation et *pool*
    - Passivation : géré par le conteneur, pas de méthodes de gestion de l’état

cf illustrations plus tard dans le cours (EJB)

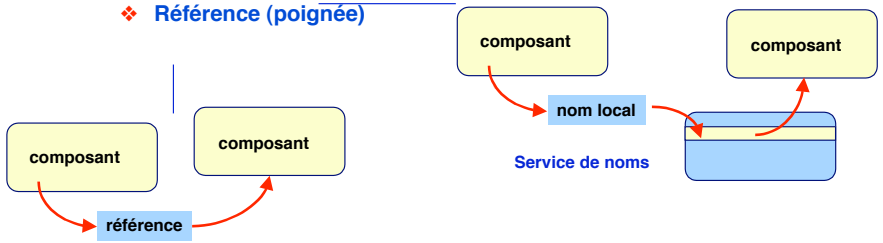
Cycle de vie des composants (3)

- Maison (*home*)
  - Partie du conteneur (visible de l'extérieur) qui gère le cycle de vie des composants qu'il inclut (du même type)
    - Engendré automatiquement (à partir d'un IDL)
  - Fonctions
    - Créer / détruire les composants : implémente *Factory*, utilise les mécanismes d'économie (*pool*, passivation)
    - Gérer les composants pendant leur existence (en particulier nommer, localiser)
  - Interface fournie
    - *create*, *find*, *remove*
    - Utilise une interface de rappel (à implémenter par le programmeur)

cf illustrations plus tard dans le cours (EJB)

Désignation des composants

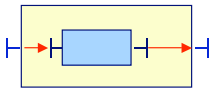
- Principe : désignation contextuelle
  - Désignation dans un contexte global
    - Service de noms (CORBA, J2EE/JNDI, …)
    - Clé pour les objets persistants (clé primaire SGBD)
  - Désignation locale pour éviter noms globaux "en dur"
    - Symbolique
    - Référence (poignée)



Liaison des composants

Les principes généraux de la liaison s'appliquent ; quelques spécificités.

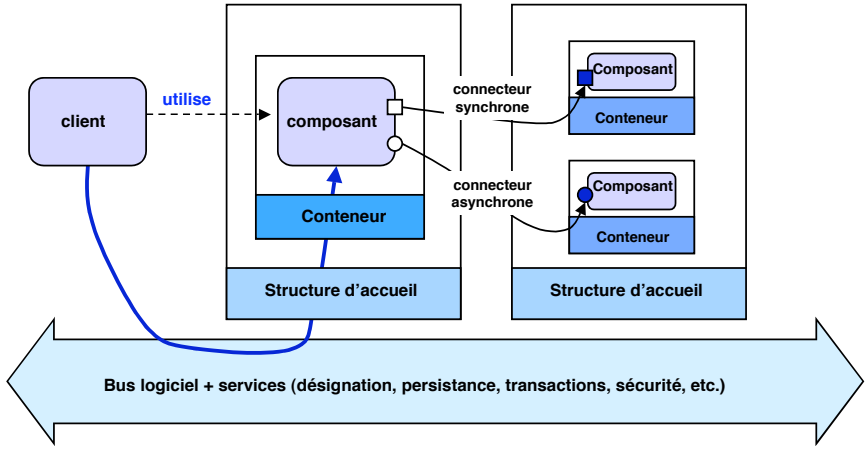
- Plusieurs types de liaison selon localité
  - Entre composants de même niveau dans un même espace : référence dans cet espace (exemple : référence Java)
  - Entre composants dans des espaces différents : objet de liaison (avec éventuellement optimisations locales)
    - Exemple EJB :
      - Dans un même conteneur : LocalObject
      - Entre deux conteneurs : RemoteObject
      - Entre client et serveur : RMI (stub + skeleton)
  - Entre composants inclus (dans les modèles qui l'autorisent)
    - Exportation - Importation



Appel d'un composant

- Principe
  - Le conteneur réalise une médiation pour l'appel d'une méthode d'un composant, via un objet intermédiaire (intercepteur)
  - Comme le reste du conteneur, l'intercepteur est engendré automatiquement (fait partie de la chaîne de liaison)
  - Le conteneur fournit en général une optimisation locale pour les appels entre composants qu'il contient
- Fonctions réalisées
  - Lien avec gestion de ressources (activation d'un composant passivé)
  - Sécurité
  - Fonctions supplémentaires "à la carte" (intercepteurs programmables)

Mise en œuvre des composants



Description d'architecture

- Ensemble des composants logiciels d'une application
  - identification des composants
  - désignation des composants
- Structuration des composants
  - composition hiérarchique (sous-composants)
- Interconnexion des composants
  - dépendances entre composants (*fournit - requiert*)
  - modes de communication (synchrone, asynchrone)
  - protocoles de communication
- Autres aspects
  - règles d'usage
  - contraintes globales (environnement requis)

Intérêt d'une description globale d'architecture

- Facilite la conception et la réalisation
  - réalise la synthèse entre
    - le cahier des charges (fonctions requises)
    - les méthodes de conception
    - la mise en œuvre
    - le déploiement en environnement réparti
- Facilite la compréhension globale du système
  - outil de documentation
- Facilite le processus d'évolution
  - modification des composants (interface, réalisation)
  - modification des relations entre composants
  - modification du déploiement

Langages de description d'architecture

MIL : *Module Interconnection Languages*
ADL : *Architecture Description Languages*

- Objectifs
  - Description globale de l'architecture d'une application
    - Définition des composants
      - interfaces fournies, requises
    - Définition des interconnexions
  - Support d'outils pour la vérification
    - Définition des types des composants
    - Vérification de conformité des interfaces
  - Support d'outils pour le déploiement
    - Annotations spécifiant le déploiement
    - Génération automatique de scripts de déploiement
- Réalisations
  - Essentiellement prototypes de recherche

Composants : situation

- La programmation par composants en est à ses débuts
  - Standards de facto côté serveur (issus de l'industrie)
    - J2EE (intègre de nombreuses technologies : EJB, JSP, MS, JNDI, JDBC,…)
    - .NET (intègre COM+, ASP.NET, C#, CLR,…)
  - Moins développé côté client
    - Exemple : JavaBeans
  - Normalisation par consortiums
    - OMG : Composants CORBA : CORBA Component Model (CCM)
    - OSGI : modèle plus léger pour applications embarquées, mobiles
  - Recherche et travaux en cours
    - Modèles et infrastructures innovants : Fractal, Avalon, … (logiciel libre)
    - Langages de description d'architectures
      - Vers des normes pour ADL ? Prototypes en cours
    - Support pour configuration et déploiement : SmartFrog, …
      - travaux en cours à l'OMG (OSD : Open Software Description)
    - Aide à la conception
      - travaux sur l'extension de l'UML

Java Beans

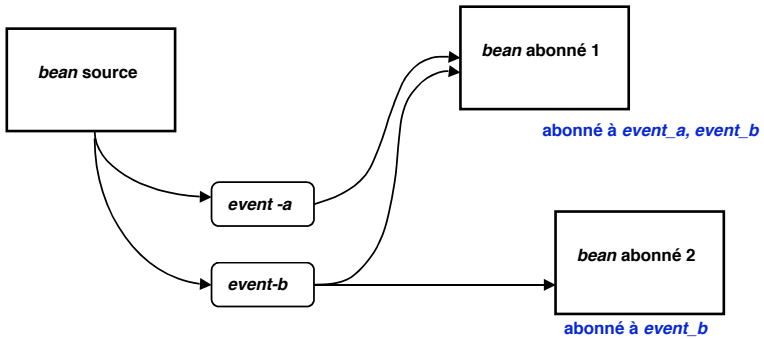
- Objectifs
  - Construction de programmes clients (en général interfaces graphiques) par composition dynamique
    - Assemblage dynamique de composants
    - Visualisation en ligne de l'application réalisée
  - Ne traite pas la répartition
- Propriétés
  - Un *bean* possède des attributs publics (exportables)
  - Un bean est configurable (par modification de ses propriétés)
  - Les *beans* communiquent entre eux par événements
  - Chaque *bean* est persistant (survit au processus créateur)
  - On peut découvrir dynamiquement les propriétés d'un *bean* (introspection)

Propriétés des *Beans*

- Définitions
  - propriété : attribut public pouvant être modifié depuis l'extérieur par d'autres *beans* ou par des outils (*PropertyEditor)*
  - droits : *read/write*, *read only*, *write only*
  - attributs de présentation (couleur, taille, etc.) ou de comportement
- Identification des propriétés
  - règles syntaxiques : `int setX(int i)` et `int getX()` identifient un attribut `x` de type entier
  - différents types : simples, indexées, liées (événement émis à chaque modification), contraintes (contrôle sur la modification via un autre *bean*)

Communication entre *Beans*

- Communication de type événement - réaction, avec abonnement préalable (*publish - subscribe*)
  - les événements sont des objets particuliers



Événements

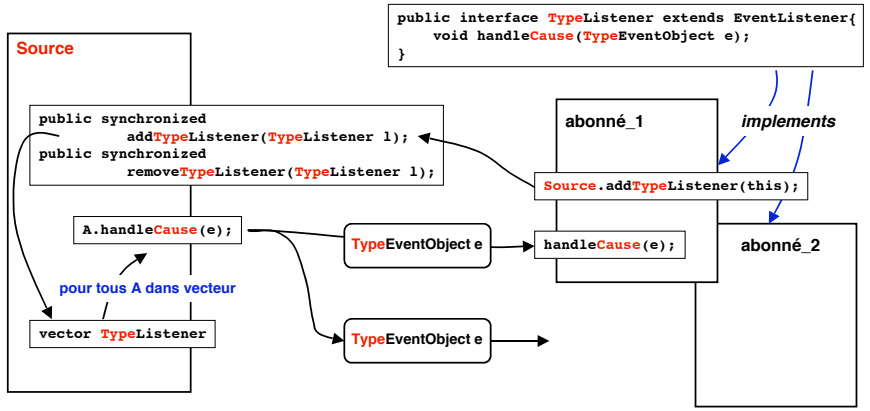
- Les événements sont des objets Java particuliers
  - héritent de la classe `Java.util.EventObject`
  - on peut définir plusieurs types d'événements dans un *bean* émetteur
- Abonnement
  - un *bean* qui doit régir à un événement doit s'abonner auprès du *bean* susceptible d'émettre cet événement, en spécifiant le type de l'événement
  - pour cela, le *bean* émetteur d'un événement de type `TypeEvent` doit fournir des méthodes d'abonnement et de désabonnement :

```
addTypeListener(TypeListener l);
removeTypeListener(TypeListener l);
```

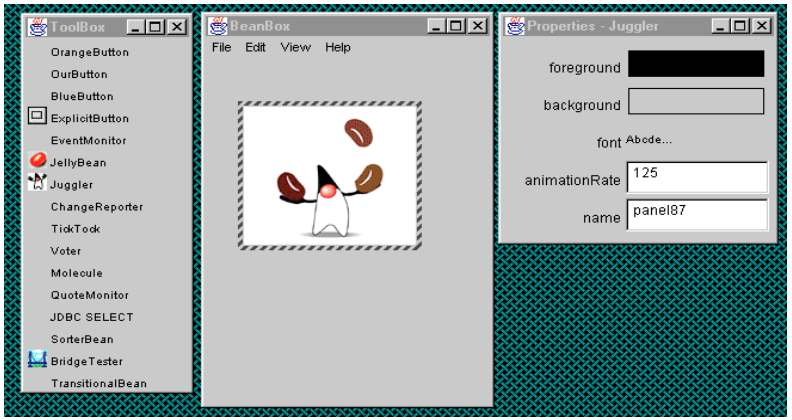
(`Type` est le type de l'événement considéré)

Événements

Le système d'événements suit un canevas bien défini (conventions de nommage et de typage)



L'environnement de développement (*BeanBox*)

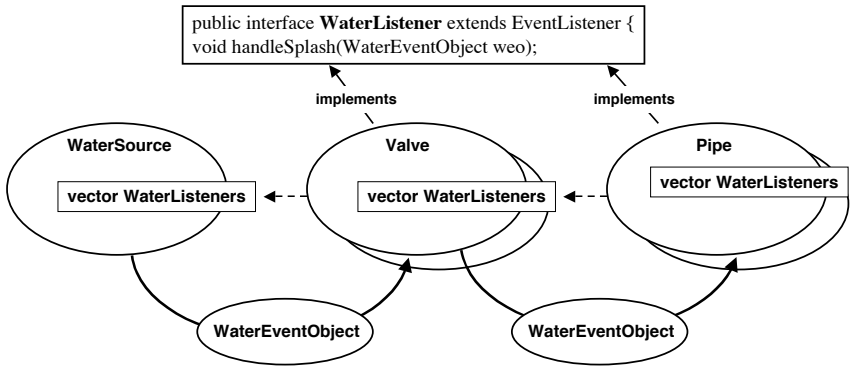


liste de *beans* espace de travail pour les *beans* propriétés des *beans*

source : http://java.sun.com/docs/books/tutoria/javabeans/

Exemple

http://java.sun.com/beans/docs/Tutorial-Nov97.pdf



La source envoie périodiquement des événements (émission d'eau) aux valves et tuyaux. Une valve peut être fermée (elle arrête l'eau) ou ouverte (elle laisse passer, donc retransmet l'événement). Un tuyau se comporte comme une valve ouverte. Un ensemble de tuyaux et valves sont interconnectés.

Exemple : définition d'un événement

WaterEventObject

```
public class WaterEventObject extends EventObject {
       long timeOfEvent;
       public WaterEventObject(Object o) {
       super(o);
       timeOfEvent = System.currentTimeMillis();
       }
public long getTimeOfEvent() {
       return timeOfEvent;
       }
}
```


Exemple : Watersource

WaterSource

```
public class WaterSource extends Canvas implements Runnable {
       private Vector waterListeners = new Vector();
       Thread thread;
       public WaterSource() {
              setBackground(Color.blue);
              thread = new Thread(this);
              thread.start();
              }
       public Dimension getMinimumSize() {
              return new Dimension(15,15);
              }
       public void run() {
              while(true) {
                     splash();
                     try {
                     thread.sleep(1000);
                     } catch (Exception e) {}
              }
       }
```

liste des *beans* qui se sont abonnés aux événements de type *waterEvent*

envoi d'eau

1 fois par seconde

Exemple : Watersource (fin)

Fin de *WaterSource*

```
       public synchronized void addWaterListener(WaterListener l) {
              waterListeners.addElement(l);
              }
       public synchronized void removeWaterListener(WaterListener l) {
              waterListeners.removeElement(l);
              }
       private void splash() {
              Vector l;
              WaterEventObject weo = new WaterEventObject(this);
              synchronized(this) {
                     l = (Vector)waterListeners.clone();
              }
              for (int i = 0; i < l.size(); i++) {
                     WaterListener wl = (WaterListener) l.elementAt(i);
                     wl.handleSplash(weo);
              }
       }
}
```

création d'unévénement WaterEvent

on envoie l'événement à la liste courante des abonnés (on en prend une copie pour éviter les interférences avec les abonnements en cours

émission d'unévénement `WaterEvent` (on active la méthode de traitement de chaque récepteur

Exemple : Valve

Valve

```
public class Valve extends Canvas implements WaterListener, Runnable {
       private Vector waterListeners = new Vector();
       private WaterEventObject lastWaterEvent;
       private boolean open = true;
       Thread thread;
       public Valve() {
              setBackground(Color.white);
              thread = new Thread(this); thread.start();
              }
       public boolean isOpen() {
              return open;
       }
       public void setOpen(boolean x) {
              open = x;
       }
       public Dimension getMinimumSize() {
              return new Dimension(20,30);
       }
       public void handleSplash(WaterEventObject e) {
              lastWaterEvent = e;
              if (isOpen()) {
                     setBackground(Color.blue);
                     repaint();
                     splash();
              }
       }
```

état ouvert ou fermé

test et manipulation de l'état

traitement de l'événement `WaterEvent`

si ouvert, propager l'événement

conventions : bleu, l'eau passe ; blanc non

Exemple : Valve (fin)

```
public void run() {
        while(true) {
        try {
                thread.sleep(1000);
        } catch (Exception e) {}
        if (lastWaterEvent != null) {
                long dt = System.currentTimeMillis() - lastWaterEvent.
                getTimeOfEvent();
                if ((dt > 2000) || (!isOpen())) {
                        setBackground(Color.white);
                        repaint();
                        }
                } }}
public synchronized void addWaterListener(WaterListener l) {
        waterListeners.addElement(l);}
public synchronized void removeWaterListener(WaterListener l) {
        waterListeners.removeElement(l);}
void splash() {
        Vector l;
        WaterEventObject weo = new WaterEventObject(this);
        synchronized(this) {
                l = (Vector)waterListeners.clone();}
        for (int i = 0; i < l.size(); i++) {
                WaterListener wl = (WaterListener) l.elementAt(i);
                wl.handleSplash(weo);}
        }
}
```

peindre la valve en blanc si un événement `WaterEvent` n'a pas été reçu dans les 2 dernières secondes, ou si la valve est fermée

même canevas que précédemment

Exemple : Pipe

Pipe

```
public class Pipe extends Canvas implements WaterListener, Runnable {
        private Vector waterListeners = new Vector();
        private WaterEventObject lastWaterEvent;
        Thread thread;

        public Pipe() {
                setBackground(Color.white);
                thread = new Thread(this);
                thread.start();
        }
        public Dimension getMinimumSize() {
                return new Dimension(150,10);
        }
        public void handleSplash(WaterEventObject e) {
                lastWaterEvent = e;
                setBackground(Color.blue);
                repaint();
                splash();
        }
```

même comportement que valve ouverte

Exemple : Pipe (fin)

```
public void run() {
      while(true) {
              try {
                      thread.sleep(1000);
              } catch (Exception e) {}
              if (lastWaterEvent != null) {
                      long dt = System.currentTimeMillis()
                              -lastWaterEvent.getTimeOfEvent();
                      if (dt > 2000) {
                              setBackground(Color.white);
                              repaint();
                      }
              }
      }
}
public synchronized void addWaterListener(WaterListener l) {
      waterListeners.addElement(l);}
public synchronized void removeWaterListener(WaterListener l) {
      waterListeners.removeElement(l);}
void splash() {
      WaterEventObject weo = new WaterEventObject(this);
      for (int i = 0; i < waterListeners.size(); i++) {
              WaterListener wl = (WaterListener)waterListeners.elementAt(i);
              wl.handleSplash(weo);
      }
}
```

peindre en blanc si un événement `WaterEvent` n'a pas été reçu dans les 2 dernières secondes (même chose que valve ouverte)

Exemple : mise en œuvre

Créer un objet *WaterSource*, et un ensemble d'objets *Pipe* et *Valve*.

Connecter les émetteurs et récepteur d'événements entre eux (l'environnement *BeanBox* permet de le faire simplement

Activer l'objet *WaterSource*

Ouvrir et fermer les valves (l'environnement permet de modifier la propriété *open* d'une valve)

On peut alors observer l'écoulement de l'eau dans le circuit (représenté en bleu ou en blanc selon que l'eau passe ou non)

Java Beans : conclusion

- **Modèle de composants**
  - intégration au niveau source (Java)
  - modèle simple et commode pour les composants individuels
  - modèle bien adapté aux composants graphiques
  - outils limités pour la description de l'architecture
    - pas de visualisation globale de l'architecture
- **Environnement de développement (*BeanBox*)**
  - outils d'aide à la construction
    - édition simple des propriétés et des méthodes
    - plus dificile pour les liens entre *Beans*
  - non prévu pour la répartition

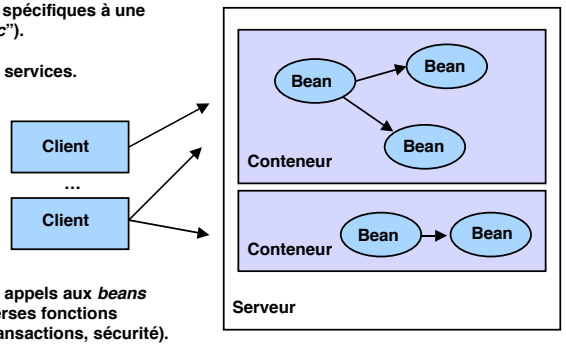
Enterprise Java Beans (EJB)

- **Objectifs**
  - Faciliter la construction des programmes pour les serveurs d'entreprise (niveau intermédiaire d'une architecture 3 niveaux) par assemblage de composants réutilisables
  - Fournir un environnement pour des services communs (persistance, transactions, etc.), permettant au développeur de se concentrer sur les problèmes spécifiques de l'application
  - Permettre le développement d'une industrie des composants en séparant les fonctions de développement de composants, d'assemblage d'applications, et de fourniture de services
- **Caractéristiques générales**
  - 2 types d'*Enterprise Beans*
    - *Entity Bean* : représente un objet, en général persistant
    - *Session Bean* : représente le déroulement d'une série d'actions pour un client
  - Environnement = serveur + conteneur

Schéma d'exécution des EJB

Les *beans* fournissent un ensemble de services, en réalisant les traitements spécifiques à une application ("*business logic*").

Les clients font appel à ces services.

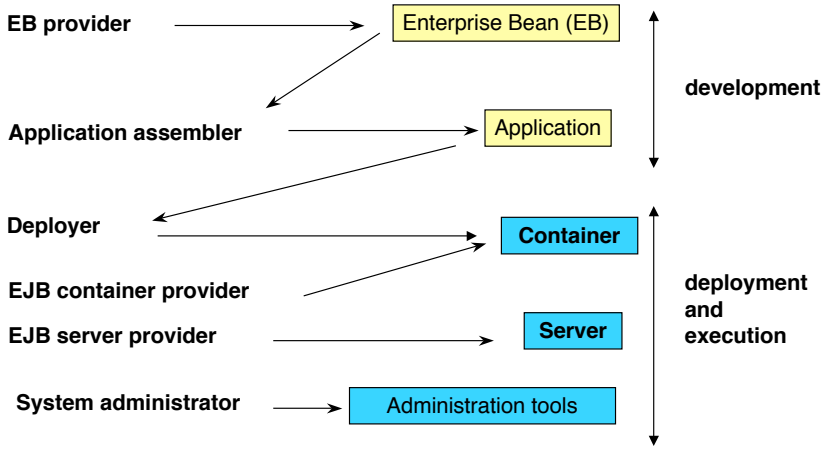


Le conteneur intercepte les appels aux *beans* qu'il contient et réalise diverses fonctions communes (persistance, transactions, sécurité).

Les conteneurs isolent les *beans* - a) des clients, - b) d'une implémentation spécifique du serveur

Le serveur joue le rôle d'intermédiaire entre le système et le conteneur (il permet au conteneur de réaliser ses fonctions en faisant appel aux primitives du système)

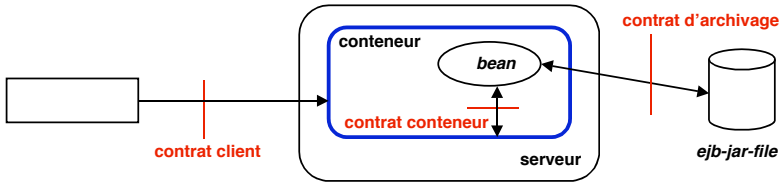
Fonctions liées aux *Enterprise Java Beans*



source : documentation JOnAS : http://www.objectweb.org/jonas/

Contrats des EJB

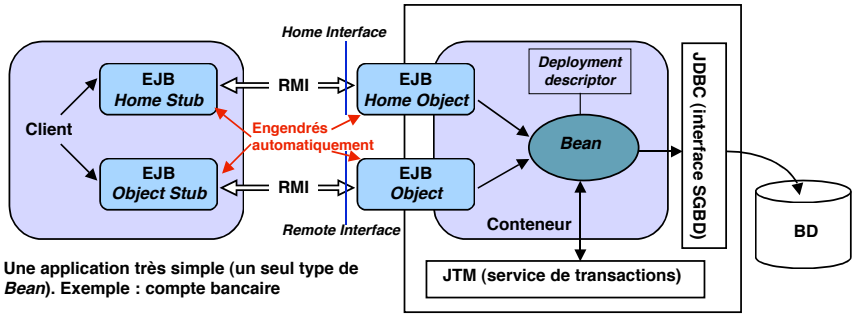
- Des "contrats" sont associés à chaque *bean*
  - doivent être respectés par le client, le *bean* et le conteneur
  - contrat côté client
    - fournit une vue uniforme du bean au client (vue indépendante de la plate-forme de déploiement)
  - contrat côté conteneur
    - permet la portabilité des *beans* sur différents serveurs
  - contrat d'archivage (*packaging*)
    - définit un format de fichier standard (*ejb-jar-file*) pour archiver les *beans*. Ce format doit être respecté par tous les outils



Contrat côté client

- Localisation du *bean*
  - utilisation de JNDI (interface Java à un service d'annuaire)
- Utilisation du *bean*
  - via les interfaces standard fournies par le développeur des *beans*
    - *Home Interface* (méthodes liées à la gestion du *bean* : *create*, *remove*, *find*, etc.
    - *Remote Interface* (méthodes propres à l'application)
- Le conteneur réalise un mécanisme de délégation
  - le client ne s'adresse pas directement au *bean*, mais au conteneur
  - le conteneur "fait suivre" les appels

Schéma d'exécution des EJB : une vue plus précise



Une application très simple (un seul type de *Bean*). Exemple : compte bancaire

À tout *Bean* sont associés un *EJB Object* et un *EJB Home Object*

L'*EJB Home Object* réalise les fonctions liées au cycle de vie du *Bean* : création, identification et recherche, destruction. Il est lui-même localisé grâce à un annuaire (interface JNDI)

L' *EJB Object* réalise l'interface d'accès aux services du *Bean*. Il intercepte tous les appels au *Bean* et assure les fonctions de transactions, persistance, gestion de l'état, sécurité, selon des modalités spécifiées dans le *deployment descriptor*.

Contrat côté conteneur

- Un conteneur réalise des fonctions pour le compte des *beans* qu'il contient
  - gestion du cycle de vie, gestion de l'état, sécurité, transactions, concurrence, etc
  - ces services appellent des méthodes fournies par le *bean* (*callback methods*)
    - exemple : *ejbCreate*, *ejbPostCreate*, *ejbLoad*, *ejbStore*, etc.
- Les conteneurs gèrent deux types de *Beans*
  - *Entity Beans* : réalisent des objets de l'application
  - *Session Beans* : réalisent des séquences d'opérations pour un client
  - Des contrats spécifiques sont définis pour chacun de ces types (avec variantes selon le degré de prise en charge des fonctions par le conteneur)

EJB *Entity Beans*

- **Propriétés**
  - représentent des entités persistantes (stockées dans un SGBD) sous forme d'objets ; les *entity beans* sont eux-mêmes persistants
    - la gestion de la persistance peut être faite par le *bean* lui même (*bean managed persistence*) ou déléguée au conteneur (*container managed persistence*)
  - partagés par plusieurs clients
    - d'où gestion de la concurrence
  - peuvent participer à des transactions
  - survivent aux arrêts où aux pannes des serveurs EJB
  - mode de création
    - création explicite d'une instance
    - insertion d'une entité dans la base de données

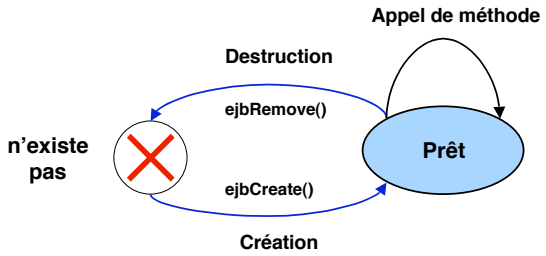
EJB *Session Beans*

- **Propriétés**
  - représentent une suite d'opérations pour le compte d'un client particulier
    - traitements en mémoire
    - accès au SGBD
  - sont créés et détruits par un client
  - non persistants
  - ne résistent pas aux pannes ou à l'arrêt du serveur
  - deux modes de gestion de l'état
    - *stateless* (sans état) : pas de données internes ; peut être partagé entre plusieurs clients ; pas de passivation (cf. plus loin)
    - *stateful* (avec état) : conserve son état sur une suite d'appels de méthodes (pour le compte d'un client et d'un seul) ; prévoir passivation et activation (cf. plus loin)

Cycle de vie des *Session Beans* (1)

Stateless Session Beans

gérés en pool par le conteneur (opérations invisibles au client)
pris dans le pool lors de l'appel de create() par le client
détruits à l'initiative du conteneur (gestion interne du pool)

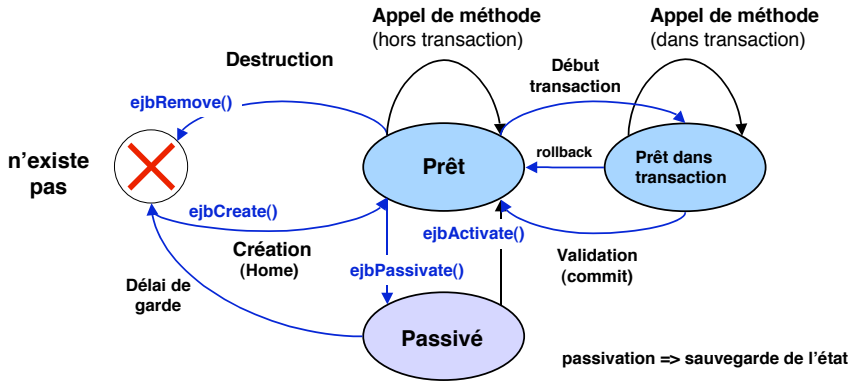


pas d'état => pas de sauvegarde

Cycle de vie des *Session Beans* (2)

Stateful Session Beans

peuvent être passivés par le conteneur si inactifs (économie de ressources)
réactivés quand le client en a besoin

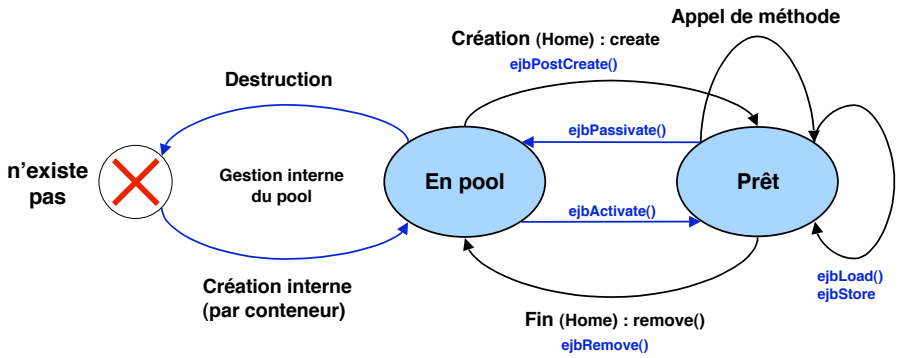


passivation => sauvegarde de l'état

Cycle de vie des *Entity Beans*

Entity Beans

création et mise en pool par le conteneur
sortie du pool après opération create sur Home par le client (+ ejbPostCreate() pour initialiser)



Gestion du cycle de vie par le conteneur

- **Administration (via *Home Interface*)**
  - Permet au client de créer, détruire, rechercher un *bean*
  - Le *bean* doit implémenter les méthodes (*callback*) correspondantes : *ejbCreate*, *ejbPostCreate* , etc
- **Gestion de l'état d'un *bean* par le conteneur**
  - Passivation
    - enregistrement de l'état du *bean* en mémoire persistante
    - désactivation du *bean* (il ne peut plus être appelé)
  - Activation
    - chargement de l'état du *bean* depuis la mémoire persistante
    - réactivation du *bean* (il redevient utilisable)
  - Le *bean* doit implémenter les méthodes (*callback*) *ejbPassivate* et *ejbActivate*

Gestion de la persistance

- **Gestion par le conteneur**
  - Le conteneur est responsable de la sauvegarde de l'état
  - Les champs à sauvegarder et le support doivent être spécifiés séparément (dans le descripteur de déploiement)
- **Gestion par le *bean***
  - Le *bean* est responsable de la sauvegarde de son état (il doit explicitement insérer les opérations d'accès permettant la gestion de la persistance dans les fonctions de *callback* appropriées
  - Moins adaptable que la gestion par le conteneur, puisque les opérations de gestion de la persistance sont insérées "en dur" dans le code

Gestion des transactions

- Conforme à l'architecture XA (X/Open Architecture)
  - en Java : Java Transaction API (JTA)
- Modèle de transaction "plat"
- Gestion déclarative des transactions
  - pour un *bean entier* ou pour chaque méthode
  - attributs transactionnels définis dans le descripteur de déploiement
  - également contrôle explicite possible par le *bean*

Attribut transactionnel associé à une méthode du *bean*	Transaction du client	Transaction exécutant la méthode du *bean*
NotSupported	--	--
	T1	--
Required	--	T2
	T1	T1
Supports	--	--
	T1	T1
RequiresNew	--	T1
	T1	T2
Mandatory	--	Erreur
	T1	T1
Never		

Gestion de la sécurité

- La gestion de la sécurité est déléguée au maximum au conteneur
  - simplifie la programmation, augmente la portabilité
- Bases de la sécurité
  - API sécurité de Java (*javax.security*)
  - méthodes liées à la sécurité dans le conteneur (*javax.ejb.EJBContext* : interface du conteneur, base de *EntityContext* et *SessionContext*)
    - *getCallerPrincipal*
    - *isCallerInRole*
  - attributs de sécurité dans le descripteur de déploiement du *Bean*
    - spécification de “rôles”
    - spécification de méthodes exécutables sous un rôle

Descripteur de déploiement

- Fonction
  - Spécification déclarative de diverses propriétés d’un *bean*
    - Identification, attributs de transaction, champs persistants, environnement, gestion ou non par le conteneur, rôles pour la sécurité, etc .
  - Utilisé par le conteneur pour mettre en œuvre ses fonctions
- Forme
  - Depuis EJB 1.1 : descripteur en XML
  - Pour chaque propriété, une balise particulière est définie par la DTD
  - Exemple

```
...
<persistence-type>Container</persistence-type>          persistance gérée par le conteneur

<container-transaction>
        <method>                                        transactions gérées par le conteneur
                <ejb-name>MyBean</ejb-name>
                <method-name>*</method-name>
        </method>                                       exécution transactionnelle obligatoire pour toutes les méthodes
        <trans-attribute>Required</ trans-attribute>
</container-transaction>
...
```


Étapes du développement d’une application en EJB (1)

- Développement d’un *Session Bean*
  - créer la *Home Interface* (étend *ejb.EJBHome*)
    - méthodes *create*, *remove*
  - créer la *Remote Interface* (étend *ejb.EJBObject*)
    - méthodes spécifiques à l’application
  - écrire l’implémentation des méthodes de création (*Create*, *PostCreate*)
  - écrire l’implémentation de l’interface
    - méthodes spécifiques
    - méthodes *callback*
      - *setSessionContext*, *ejbActivate*, *ejBPassivate*, etc
      - si le *bean* a un état géré par le conteneur : *afterBegin*, *afterCompletion*, *beforeCompletion*, etc

Étapes du développement d’une application en EJB (2)

- Développement d’un *Entity Bean*
  - créer la *Home Interface* (étend *ejb.EJBHome*)
    - méthodes *create*, *remove*
  - créer la *Remote Interface* (étend *ejb.EJBObject*)
    - méthodes spécifiques à l’application
  - écrire une classe sérialisable “clé primaire” (*nomduBeanPK*) destinée à servir de clé de recherche et de manipulation du *bean*
  - écrire l’implémentation des méthodes de création (*Create*, *PostCreate*)
  - écrire l’implémentation de l’interface
    - méthodes spécifiques
    - méthodes *callback*
      - *setEntityContext*, *ejbActivate*, *ejBPassivate*, etc
      - *ejbLoad*, *ejbStore*, etc. Si la persistance est gérée par le conteneur, *ejbLoad* et *ejbStore* peuvent être vides.

Étapes du développement d'une application en EJB (3)

- **Écrire un descripteur de déploiement**
  - un par *bean*
  - définit le comportement pour les transactions, la persistance, la sécurité, le lien avec les bases de données utilisées, l'environnement (placement sur les serveurs), etc.
- **Mettre en place le serveur**
  - compiler les programmes des *beans*
  - engendrer les classes du conteneur (implémentation des interfaces *Home* et *Remote*) au moyen de l'outil adéquat (GenIC dans JOnAS) auquel on fournit les classes des *beans* et le descripteur de déploiement
- **Développer et lancer le programme client**
  - le client obtient les références aux *beans* via un service de noms
  - le client peut créer et lancer des sessions, ou appeler directement les *Entity beans*

Exemples

Voir exemple dans la documentation de JOnAS : http://objectweb.org/jonas/

Exemple simple : serveur de comptes bancaires

un *Entity Bean* en deux versions
- *AccountImpl* : persistance implicite(gérée par le conteneur)
- *AccountExpl* : persistance explicite(gérée par le *bean*)

un programme client (sans session) *ClientAccount*
un descripteur de déploiement *ejb-jar*
un fichier *Makefile*

Exemple plus complexe : serveur de commerce électronique

ECOM
RICOM

Conclusion sur les EJB (pour l'utilisateur)

- **Un modèle à base de composants pour les serveurs**
  - largement adopté ; influence la normalisation (OMG)
- **Avantages**
  - simplifie la réalisation d'applications complexe en libérant le concepteur des aspects non directement liés à l'application
    - gestion déclarative des transactions
    - gestion de la persistance
    - gestion de la sécurité
    - gestion de la répartition
  - augmente l'indépendance entre plate-forme et applications
    - séparation des rôles des fournisseurs
    - ouverture, concurrence, amélioration de la qualité
  - modèle extensible

Références sur les EJB

Références générales

R. Monson-Haefel, *Enterprise Java Beans*, 3rd edition, O'Reilly, 2000

http://java.sun.com/products/ejb/

Produits

JOnAs (Bull-ObjectWeb) : logiciel libre : http://www.objectweb.org/jonas/

JBoss : logiciel libre + services payants : http://www.jboss.com/

WebLogic (BEA) : http://www.weblogic.com/

WebSphere (IBM) : http://www.ibm.com/software/websphere/

Fonctionnement interne d'un serveur EJB

M. Völter, A. Schmid, E. Wolff, *Server Component Patterns*, Wiley 2002
Voir aussi http://www.voelter.de/conferences/tutorials.html